# Module 8: Wrirting Loops and Functions in R

The purpose of this handout is to introduce you to using R for automate repetitive tasks by writing for loops and custom functions. You do not need to have completed any other Modules in order for this Module to be useful.

This handout will cover the following analyses:

- `for` loops
- Writing your own functions

## 1 Understanding Functions

It may be obvious, but nearly everything you do in R is done by a function. Want to read in your data? Use the `read.csv()` function. Want to calculate a mean? Use the `mean()` function. Want to run a generalized linear mixed effects model? Better use the `glmer()` function.

Functions are sets of commands that are bundled together in order to accomplish some task. Generally, but not always, functions requre some input(s) and return some sort of output(s). This can be as simple as the `mean()` function, which simply requies a vector of numbers, to something like a complex statistical model like we made in Module 7. Writing functions and loops requires a little bit of an open mind and an ablity to think in the abstract.

Although R may seem like it has a function for every possible task, it does not. Thus, there are many reasons why you might want to create your own functions. For example, you might want to write a function for any of the following tasks:

- Editing data
- Repeating similar analyses on different variables
- Creating similar graphs from different variables
- Creating customized graphs that require a lot of code
- Running simulations
- Lots of other reasons where you are doing something over and over

## 2 `for` loops

Probably the simplest way to automate completing some task is to use a `for` loop, which is simply a recursive command that specfies a task (or tasks) to be completed a set number of times. Essentially, a `for` loop is as follows:

```
for(i in *vector of number of times to do task*)
{
*task to be completed*
print(*return some output to the console*)
}
```

For example, if you wanted to make a list of plants in your study (by number) you could easily use a `for` loop to generate it very quickly. The important things to note in the code below are as follows: 1) we have defined a variable called `i` and set it equal to 1 through 20, so the action happened 20 times. Note that `i` could be called anything (it does not have to be `i`). 2) we had to include the command `print` to make sure that at the end of each loop the output was printed to the console.

```
> for(i in 1:20)
+ {
+  print(paste("plant",i))
+ }
[1] "plant 1"
[1] "plant 2"
[1] "plant 3"
[1] "plant 4"
[1] "plant 5"
[1] "plant 6"
[1] "plant 7"
[1] "plant 8"
[1] "plant 9"
[1] "plant 10"
[1] "plant 11"
[1] "plant 12"
[1] "plant 13"
[1] "plant 14"
[1] "plant 15"
[1] "plant 16"
[1] "plant 17"
[1] "plant 18"
[1] "plant 19"
[1] "plant 20"
```

So what *exactly* does the code above do? First, we set `i` equal to 1. Second, we execute a function called `paste()` which simply pastes two bits of text together. Here, we pasted together the word "plant" and whatever `i` is currently set at. Third, the output of the `paste()` function is printed to the console. When the loop gets to the end of executing those three things, it sets `i` equal to 2 and repeats everything, and so on, until the end of our list of values for `i`.

It is often useful to not have to define everything exactly, but instead to simply define the number of loops to run based on some other pre-existing variable. For example, in the RxP study (Modules 2-4) we were studying aspects of metamorphosis in amphibians. Without even knowing how many observations there were, we can define a `for` loop that runs for as long as our data frame is. Below we have set the `for` loop to run for as many rows as are in the RxP.clean data frame. Another advantage of this method is that if we decide to change our data at all (e.g., remove some outliers), we can rerun this code without modifying a thing.

```
> for(i in 1:nrow(RxP.clean))
{
print(paste("I am tadpole",i))
```

```
}
[1] "I am tadpole 1"
[1] "I am tadpole 2"
[1] "I am tadpole 3"
[1] "I am tadpole 4"
[1] "I am tadpole 5"
[1] "I am tadpole 6"
[1] "I am tadpole 7"
[1] "I am tadpole 8"
...
[1] "I am tadpole 2492"
[1] "I am tadpole 2493"
```

We can also use `for` loops to do more useful but repetitive tasks. For example, maybe we want see if the effects of predators on metamorph SVL differ across our 8 spatial blocks in the RxP experiment. We can use a `for` loop to conduct a model on each block in succession.

```
for(i in 1:length(unique(RxP.clean$Block)))
{
mod1<-lm(log.SVL.final~Pred, data=RxP.clean[RxP.clean$Block==i,])
print(anova(mod1))
}
```

Let's break down what the code above is doing. First, we defined the length of the loop from 1 until however long the vector of unique values of `RxP.clean$Block` is. `unique` is a function that looks at a vector and figures out how many different (i.e. unique) things are in it. In each iteration of the loop we create a linear model that evaluates the effect of predators on the log-transformed metamorph SVL. We could, of course, have just set `i` to be 1 through 8, since we know how many blocks there are, but this builds in more generality, which can be a good thing. In each step, we've used square brqackets to subset the data so that `RxP.clean$Block` is equal to `i`. Thus, when `i` is 1, the model will only use data from block 1, when `i` is 2 it will run for block 2, and so on. In each case we assign our model to be called `mod1` and then at the end of each step, we print the `Anova()` output (from the `car` package) summarizing the model. Note of course that if you try to run this without having the `car` package loaded, you will get a series of errors.

```
> for(i in 1:length(unique(RxP.clean$Block)))
+ {
+  mod1<-lm(log.SVL.final~Pred, data=RxP.clean[RxP.clean$Block==i,])
+  print(Anova(mod1))
+ }
Anova Table (Type II tests)

Response: log.SVL.final
           Sum Sq  Df F value    Pr(>F)
Pred      0.27219   2  19.641 7.036e-09 ***
Residuals 2.89640 418
---
```

```
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
Anova Table (Type II tests)

Response: log.SVL.final
          Sum Sq  Df F value    Pr(>F)
Pred      0.1916   2  11.479 1.409e-05 ***
Residuals 3.4301 411
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
Anova Table (Type II tests)

Response: log.SVL.final
           Sum Sq  Df F value    Pr(>F)
Pred      0.34603   2  22.762 5.924e-10 ***
Residuals 2.37156 312
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
Anova Table (Type II tests)

Response: log.SVL.final
           Sum Sq  Df F value    Pr(>F)
Pred      0.19132   1  23.867 1.934e-06 ***
Residuals 1.84370 230
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
Anova Table (Type II tests)

Response: log.SVL.final
           Sum Sq  Df F value    Pr(>F)
Pred      0.24824   2  18.089 3.521e-08 ***
Residuals 2.25743 329
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
Anova Table (Type II tests)

Response: log.SVL.final
           Sum Sq  Df F value    Pr(>F)
Pred      0.33064   1  48.984 2.309e-11 ***
Residuals 1.70773 253
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
Anova Table (Type II tests)

Response: log.SVL.final
           Sum Sq  Df F value    Pr(>F)
Pred      0.90888   1  103.26 < 2.2e-16 ***
```

```
Residuals 2.42054 275
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
Anova Table (Type II tests)

Response: log.SVL.final
           Sum Sq  Df F value    Pr(>F)
Pred      0.51877   1  54.823 2.114e-12 ***
Residuals 2.31832 245
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1
```

That is a little unwieldy don't you think? It's kind of a messy way to examine those results. What if instead we just wanted to get the p-value from the `Anova()` output? Just like any other object, we can examine the structure of the `Anova()` output.

```
> str(Anova(mod1))
Classes anova and 'data.frame': 2 obs. of  4 variables:
 $ Sum Sq : num  0.519 2.318
 $ Df     : num  1 245
 $ F value: num  54.8 NA
 $ Pr(>F) : num  2.11e-12 NA
 - attr(*, "heading")= chr  "Anova Table (Type II tests)\n" "Response: log.SVL.final"
```

Here we can see that the p-value is stored in an object within the `Anova` output called `$Pr(>F)`. We can access that with the `$` operator. We can also see that there are two objects within `$Pr(>F)`, but we do not want the second one (which is just an NA). Also, as with many things in `R`, you do not even have to write the whole name of the `$Pr(>F)` object, just the first few letters will suffice.

```
> Anova(mod1)$Pr[1]
[1] 2.114245e-12
```

Okay, now that we know how to access just the p-value, we can run a `for` loop to extract them and store them in table form for us. First, let's create an empty data frame to store the values. It is useful to create the data frame from a matrix, which allows us to specify the exact size of the data frame and to make it empty at the beginning. Then we'll run the loop to 1) make a useful name in the first column of the table and 2) place the p-value in the second column.

```
> block.pred.effects<-data.frame(matrix(nrow=8, ncol=2, data=NA))
> names(block.pred.effects)<-c("Block","pval")
> for(i in 1:length(unique(RxP.clean$Block)))
+ {
+ block.pred.effects[i,1]<-paste("Block",i)
+ mod1<-lm(log.SVL.final~Pred, data=RxP.clean[RxP.clean$Block==i,])
+ block.pred.effects[i,2]<-Anova(mod1)$Pr[1]
+ }
> block.pred.effects
    Block         pval
```

```
1 Block 1 7.036233e-09
2 Block 2 1.409241e-05
3 Block 3 5.924108e-10
4 Block 4 1.933793e-06
5 Block 5 3.521284e-08
6 Block 6 2.308595e-11
7 Block 7 8.356481e-21
8 Block 8 2.114245e-12
```

It looks like there are indeed significant effects of predators across each of the blocks in our experiment. Whew, that is a relief!

# 3 Writing Functions

Functions are similar to loops in that they are very useful for automating repetitive tasks, but they are also considerably different. Loops execute a specific task many times. They are essentially the same as if you just typed out the commands over and over. Thus, objects that are created within loops are stored in the memory. In the example above, the `mod1` object will be found in your memory and will be the last version of it that the loop created. In functions, objects that are created internally to the function *never exist outside the function*, unless you specify them to. Thus, you can have a very complex function that defines many temporary objects, but those objects will not exist outside the function and therefore will not clutter up your workspace.

The other major difference is that functions are generally designed to have flexible input, so you can have a single function that you can do multiple things with. This all depends of course on how you build your function and what arguments you build in to it.

## 3.1 General advice about writing functions

When you want to write a function, it is a good idea to sit down with pen and paper and think about what you want the function to do. It is also helpful to think of it as an iterative process. Start with a very simple function and build from there. One major benefit of this is it helps you get any kinks worked out as you expand your function and it grows in complexity. It is also a very good idea to very thoroughly annotate your functions with descriptions of what different lines are doing.

In the most basic form, a function looks more or less like this:

```
function_name = function(argument1, argument2)
{
  ##annotation goes here
  commands go here
  return(output goes here)
}
```

Notice that instead of ending with `print()` like in the loop, we end with `return()`. Whatever you specify in the `return()` function is what the function will output. As mentioned above, any other variables or objects you define will not exist outside of the function.

## 3.2 Exploring block effects

Let's begin by creatiung a function to do what our for loop did above. We are starting small and will build from here, so if this seems unnecessary, please bare with me.

We start by giving our function a name and defining what *arguments* we will pass it. Below, I called the function `block_effects` and have said there is going to be one argument called `dataset`.

```
block_effects<-function(dataset)
{
  #Create a blank dataframe and give it useful column names
  output<-data.frame(matrix(nrow=8, ncol=2, data=NA))
  names(output)<-c("Block","pval")
  #Run the for loop to calculate the p-value for each block
  for(i in 1:length(unique(dataset$Block)))
  {
    output[i,1]<-paste("Block",i)
    mod1<-lm(log.SVL.final~Pred, data=dataset[dataset$Block==i,])
    output[i,2]<-Anova(mod1)$Pr[1]
  }
  #return the output
  return(output)
}
```

That function should just do everything our for loop did. The argument provides what the datset is, but everything else is fixed. In the specification of `mod1` we have merely changed the name of our data from `RxP.clean` to `dataset`, which is of course defined as `RxP.clean` for the life of the function. Let's see if it worked.

```
> block_effects(RxP.clean)
    Block         pval
1 Block 1 7.036233e-09
2 Block 2 1.409241e-05
3 Block 3 5.924108e-10
4 Block 4 1.933793e-06
5 Block 5 3.521284e-08
6 Block 6 2.308595e-11
7 Block 7 8.356481e-21
8 Block 8 2.114245e-12
```

Sure enough, it did what we wanted it to. Note that just like any other function, if you want to save the output you need to assign it to an object.

```
> temp<-block_effects(RxP.clean)
> temp
    Block         pval
1 Block 1 7.036233e-09
2 Block 2 1.409241e-05
3 Block 3 5.924108e-10
```

```
4 Block 4 1.933793e-06
5 Block 5 3.521284e-08
6 Block 6 2.308595e-11
7 Block 7 8.356481e-21
8 Block 8 2.114245e-12
```

Okay, now we are ready to really build in a little more utility in to our function. Let's redefine `block_effects` to include more arguments that specify the predictor and response variables.

```
block_effects<-function(response,predictor,dataset)
{
  #Create a blank dataframe and give it useful column names
  output<-data.frame(matrix(nrow=8, ncol=2, data=NA))
  names(output)<-c("Block","pval")
  model_specs<-paste(response, predictor, sep="~")
  #Run the for loop to calculate the p-value for each block
  for(i in 1:length(unique(dataset$Block)))
  {
    output[i,1]<-paste("Block",i)
    mod1<-lm(formula(model_specs), data=dataset[dataset$Block==i,])
    output[i,2]<-Anova(mod1)$Pr[1]
  }
  #return the output
  return(output)
}
```

Here is a basic rundown of what that code does. It's the same as our for loop, but now we have the ability to name what response and predictor variables we want to use. Due to the vagaries of coding languages, this may not be exactly as straightforward as you might expect. First, we have specify our variables in quotation marks (see example below). Second, we have to create an object that will hold our model specifications which can then be read and turned into our model by the function `formula()`.

Now, we can easily look at different predictors and response variables across the block of our experiment. For example, maybe we want to look at effects of all three treatment variables on final Mass at metamorphosis.

```
> block_effects("Mass.final", "Pred", RxP.clean)
    Block         pval
1 Block 1 1.251007e-08
2 Block 2 1.220557e-04
3 Block 3 5.844520e-08
4 Block 4 3.696829e-08
5 Block 5 3.187146e-05
6 Block 6 1.095389e-10
7 Block 7 1.060584e-19
8 Block 8 9.069668e-10
> block_effects("Mass.final", "Res", RxP.clean)
    Block         pval
```

```
1 Block 1 1.090940e-05
2 Block 2 1.661235e-06
3 Block 3 9.431698e-07
4 Block 4 2.409539e-02
5 Block 5 5.005760e-03
6 Block 6 1.892404e-01
7 Block 7 7.965091e-09
8 Block 8 1.806268e-01
> block_effects("Mass.final", "Hatch", RxP.clean)
    Block         pval
1 Block 1 0.7863200950
2 Block 2 0.0005512383
3 Block 3 0.9616305524
4 Block 4 0.0005643797
5 Block 5 0.0011956966
6 Block 6 0.9128489307
7 Block 7 0.8636298467
8 Block 8 0.0472639219
```

# 4 Take home message

Hopefully this has helped to somewhat demystify writing loops and functions and demonstrate that it really is not that hard. If we wanted, we could use our `block_effects` function to build another function that takes a list of predictors and a list of responses and would run all the models and spit it out in a single table for us to interpret. Wouldn't that be something?!

Remember:

- Start small and build up from there. If you just try to dive right in to a big complex function you are more likely to make mistakes and get very frustrated.
- Taking the time to plan and think carefully about what you want your function to do can be invaluable. Think about what you need the function to do and walk through how you will do it. Do you need to make a data frame or matrix to store values? Do you need to use a `for` loop to cycle through something?
- Don't forget to include the `return()` function at the end.

# 5 Assignment!

This weeks assignment is super simple. Show me how each of the three Treatment variables affects the time to resorb the tail (`Resorb.days`) across each of the 8 blocks.